Kevin Weng

Professor Justin Tojeira

CSCI 335

5/10/24

	1st runtime	2nd runtime	3rd runtime
std::sort	82 ms	5484 ms	158326 ms
quickSelect1	498 ms	827449 ms	254886561 ms
quickSelect2	155 ms	7404 ms	159803 ms
countingSort	1401 ms	3588 ms	806458 ms
countingSort unique values	787	3588	4797

Aglorithm Analysis:

Std::sort:

The overall time complexity for std::sort is O(n log n) where n is the number of elements in the array. In this function, I calculated the minimum and maximum values and 25th percentile, median and 75th percentile. I got the minimum value by accessing the first index of the array and the maximum by accessing the last index of the array. For the 25th percentile, I multiplied by 0.25, for median I multiplied the array size by 0.50 and for the 75th percentile, I multiplied the array size by 0.75. When sorted, the best case for finding the minimum and maximum is O(1).

The run time is faster when tested with less values but with more values, it takes longer.

QuickSelect1:

The worst case time complexity for quickSelect1 is O(n^2) and the average case is O(n). The partition function has a runtime of O(n) where n is the number of elements in subarray that is being partitioned. It iterates through the elements and performs the swapping. For insertionSort, the time complexity is O(n^2) where n is the number of elements being sorted. It's time complexity is O(n^2) because each element in the subarray that is being sorted must be compared and may be swapped. QuickSelect1 is the slowest out of all other algorithms as the input size increases.

QuickSelect2:

The worst case time complexity for quickSelect2 is O(n^2) and the average case is O(n). The other function's runtime is basically the same as quickSelect1 since we reused majority of the code we had for quickSelect1. QuickSelect2 although is a little faster than QuickSelect1 in terms of different input size but still not that fast.

CountingSort:

For countingSort, we had to create a hash map and import the numbers onto the hash table. Creating and importing the numbers onto the hash table has a time complexity of O(n) because you're going through the whole table one by one to insert each number. We then needed a vector where we keep the pairs together, the value and the number of time the number repeats itself. The time complexity when sorting the vector is O(n log n). After everything is sorted, finding the percentiles has a time complexity of O(n). Overall, the time complexity of countingSort is O(n + k) where n is the number of elements in the input array and k is the range of the input numbers. When you have less unique values and more copies of each value, the runtime is faster compared to having more unique values and less copies of each value.